Comunicación interprocesos

Al ejecutarse de manera concurrente los procesos en un S.O. pueden ser independientes o procesos que cooperan entre sí. Un proceso es independiente si no afecta a otros procesos o si no es afectado por estos. Cualquier proceso que no comparta datos con otros es independiente. Por otro lado, un proceso coopera si afecta o puede ser afectado por otros procesos que estén ejecutándose en el sistema. Además, cualquier proceso que comparta datos es un proceso que coopera. Los sistemas operativos proporcionan un entorno que permite la cooperación entre procesos. Algunas razones son:

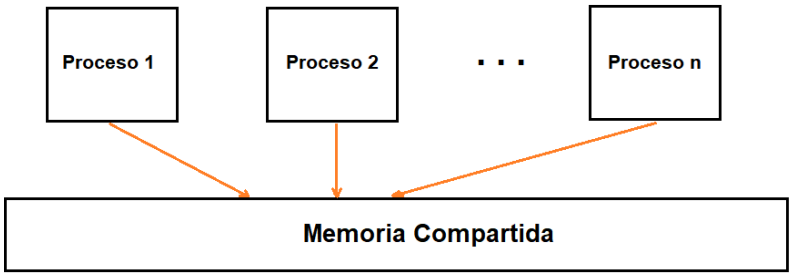
- Compartir información. Varios procesos pueden necesitar la misma información dentro de un archivo, así que es necesario tener un medio para hacerlo.
- Acelerar el tiempo de ejecución. Una tarea compleja puede dividirse en varias subtareas, las cuales, pueden ser asignadas a diversos procesos. Estos pueden ejecutarse en paralelo en una computadora con varios núcleos.
- Un usuario puede trabajar en varias tareas al mismo tiempo.
- Modularidad. Permite construir un sistema de manera modular dividiéndolo en funciones que realiza cada proceso por separado.

Para que los procesos cooperen entre sí requieren de un mecanismo que les permita intercambiar datos e información. Este mecanismo es la comunicación interprocesos *(IPC – Inter Process Communication).* Principalmente, existen dos modelos de IPC:

- Modelo de memoria compartida. Se define una región de memoria que es compartida por los procesos que cooperan entre sí.
- Modelo de paso de mensajes. Los procesos pueden intercambiar mensajes directamente entre ellos.

Memoria compartida

Este modelo permite que dos o más procesos compartan un segmento de memoria y los datos alojados ahí. Cuando creamos nuevos procesos con fork() tanto el padre como sus hijos tienen espacios en direcciones separadas. Esto de alguna manera es seguro ya que estos procesos no interfieren entre sí. No comparten nada y en principio no tienen forma de comunicarse entre ellos. Por otro lado, cuando los procesos comparten un segmento de memoria a la cual están atados, los procesos cooperarán entre ellos ya que tienen acceso a los mismos espacios de direcciones.



Memoria compartida en Linux

En Linux creamos un segmento de memoria compartida con la llamada al sistema shmget(). El propietario del segmento puede asignar propiedad, sobre ese mismo segmento, a otros procesos usando la llamada shmctl(). El propietario original también puede revocarla cuando sea necesario. Otros procesos pueden ejecutar varias funciones de control, si tienen los permisos apropiados, con la llamada shmctl(). Una vez que el segmento ha sido atado a algún proceso, éste puede leer o escribir sobre él, siempre y cuando tenga el permiso necesario. Las siguientes llamadas permiten el manejo de memoria compartida:

- **shmget()** permite obtener acceso a un segmento de memoria compartida.

  int shmget(key_t key, size_t size, int shmflag)

  key es el valor numérico que se asigna al segmento de memoria
  size es el tamaño en bytes de la memoria compartida
  shmflag especifica los permisos de acceso inicial y la creación de bandera de control, los siguientes son los valores permitidos:
  - IPC_CREAT: Para crear un nuevo segmento. Si no se declara, entonces shmget() buscará un segmento con la clave *key* y comprobará si el proceso tiene los permisos para acceder a él.
  - mode_flags: establecen los permisos de acceso

propietario	grupo	otros
r w x	r w x	r w x

  Si la llamada tiene éxito devolverá el ID del segmento de memoria compartida. En otro caso devolverá un número negativo.

- **shmctl()** es usada para modificar los permisos y otras características de la memoria compartida.

  int shmctl(int shmid, int cmd, struct shmid_ds *buf);

  Para poder usar esta llamada se debe contar con el valor *shmid*. El argumento *cmd* es uno de los siguientes comandos de control:

  - SHM_LOCK bloquea el segmento de memoria. Se debe contar con el ID del super usuario.
  - SHM_UNLOCK desbloquea el segmento de memoria y también se necesita el ID de super usuario.
  - IPC_STAT devuelve el estado de la estructura de control de la memoria asociada a shmid y lo devuelve por medio de la estructura de datos a donde**buf* apunta.
  - IPC_SET establece permisos de acceso. Se debe contar con el ID correspondiente.
  - IPC_RMID remueve el segmento de memoria compartida.
- **shmat() y shmdt()** respectivamente adjuntan y desligan segmentos de memoria compartida.

  void *shmat(int shmid, const void *shmaddr, int shmflg);

  int shmdt(const void *shmaddr);

  shmat() devuelve un apuntador *shmaddr* a la cabeza del segmento asociado que tenga un *shmid* válido.

Ejemplo: los dos programas siguientes representan un cliente y un servidor que utilizan una porción de memoria compartida. Sobre la memoria compartida escriben una cadena de caracteres.

servidor.c

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

#define TAMANIO 27

void main(){
        char c;
        int shmid;
        key_t key;
        char *shm, *s;

        key = 56789;  // Nombramos el segmento de memoria compartida
```

```
        // Creamos el segmento
        if( (shmid = shmget(key, TAMANIO, IPC_CREAT|0666)) < 0){
                perror("Error al crear el segmento de memoria");
                exit(1);
        }

        // Ligamos el segmento al espacio de direcciones
        if( (shm = shmat(shmid, NULL, 0)) == (char *)-1){
                perror("Error al ligar la memoria");
                exit(1);
        }

        // pasamos a s el apuntador a la memoria compartida
        s = shm;

        // Ponemos información sobre la memoria compartida
        for(c = 'a'; c <= 'z'; c++)
                *s++ = c;

        /*
        ponemos en espera el servidor hasta que el proceso
        cliente cambie el primer caracter de la memoria por
        '*', lo cual, indicará que ha leído la información que
        el servidor puso en la memoria
        */
        while(*shm != '*')
                sleep(1);
        exit(0);

}
```

cliente.c

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#define TAMANIO 27

void main(){
        int shmid;
        key_t key;
        char *shm, *s;
        key = 56789;

        // Localizamos el segmento de memoria
        if( (shmid = shmget(key, TAMANIO, 0666)) < 0){
                perror("No se encontró el segmento de memoria");
                exit(1);
        }

        // Ligamos el segmento
        if( (shm = shmat(shmid, NULL, 0)) == (char *) -1){
                perror("Error al ligar el segmento");
                exit(1);
        }

        // Se leen los caracteres puestos por el servidor en la memoria compartida
        for(s = shm;  *s != '\0'; s++)
                putchar(*s);
        putchar('\n');

        // Cambiamos el primer caracter por '*'
        *shm = '*';
        exit(0);

}
```

Condición de carrera

Cuando escribimos código en donde dos o más procesos comparten memoria, por ejemplo, para almacenar una variable global (en un entorno multiprogramación) es posible que ocurran errores. Al código que accede o modifica un recurso compartido se le llama **sección crítica.** Ejemplo. Supongamos que tenemos dos procesos que comparten la variable global "resultado", y tienen las siguientes instrucciones:

Proceso 1:

A1. Carga el valor de resultado al registro correspondiente en el CPU: reg1 ← resultado
A2. Suma 3 al valor del registro: reg1 ← reg1 + 3
A3. Cargar el nuevo valor en resultado: resultado ← reg1

Proceso 2:

B1. Carga el valor de resultado al registro correspondiente en el CPU: reg2 ← resultado
B2. Resta 3 al valor del registro: re2 ← reg2 – 3
B3. Cargar el nuevo valor en resultado: resultado ← reg2

De manera ideal cada proceso realizaría sus tres operaciones y el resultado final siempre sería correcto. Pero puede darse el caso siguiente: el proceso 1 ejecuta A1 y A2. En ese momento el proceso 1 se interrumpe y detiene la ejecución, y el resultado de la variable resultado es guardado en otro arreglo temporal hasta que se reinicie el proceso 1. Después proceso 2 ejecuta B1, B2 y B3. Finalmente, el proceso 1 retoma su ejecución justo donde se quedó y realiza A3.

Si el valor de resultado, antes de ejecutar Proceso 1 y Proceso 2, es 4, esperaríamos de manera ideal que fuese 4 después de ejecutar los dos procesos. Pero no es así ya que al ser interrumpido Proceso 1se guardo en su estado el valor del registro en ese momento reg1 = 7. Una vez que Proceso 1 retomó su ejecución el valor del registro para la variable resultado será 7, sin importar lo que haya hecho antes Proceso 2. A esta situación se le conoce como *condición de carrera o race condition.*

Las condiciones de carrera son problemas potenciales en IPC cuando dos o más procesos interactúan por medio de datos compartidos. El resultado final depende de la secuencia de instrucciones exacta de los procesos.

Ejemplo. Tres programas, uno produce procesos que restan a la variable compartida una unidad. El restador también la inicializa. El sumador suma una unidad, y el tercer proceso solo muestra en pantalla el valor de la variable compartida.

```
#restador.c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#define TAMANIO 1
void main(){
        int shmid; // ID del segmento de memoria compartida
        key_t key;
        int *shm, *s;

        key = 56789;  // Nombramos el segmento de memoria compartida

        // Creamos el segmento
        if( (shmid = shmget(key, TAMANIO, IPC_CREAT|0666)) < 0){
                perror("Error al crear el segmento de memoria");
                exit(1);
        }

        // Ligamos el segmento al espacio de direcciones
        if( (shm = shmat(shmid, NULL, 0)) == (int *)-1){
                perror("Error al ligar la memoria");
                exit(1);
        }

        s = shm;
        *s = 0; // valor inicial de la variable compartida

        fork();
        fork();
        fork();
        fork();
        fork();
        fork();
```

```
        fork();
        fork();
        fork();
        fork();
        fork();

        int i = 0;
        while(i < 10){
                *s = *s-1;
                printf("\nValor de variable en restador = %d",*s);
                //sleep(4);
                i++;
        }
}
```

```
#sumador.c
:
#define TAMANIO 1
void main(){




        int shmid; // ID del segmento de memoria compartida
        key_t key;
        int *shm, *s;


        key = 56789;  // Nombramos el segmento de memoria compartida

        // Creamos el segmento
        if( (shmid = shmget(key, TAMANIO, 0666)) < 0){
                perror("Error al crear el segmento de memoria");
                exit(1);
        }

        // Ligamos el segmento al espacio de direcciones
        if( (shm = shmat(shmid, NULL, 0)) == (int *)-1){
                perror("Error al ligar la memoria");
                exit(1);
        }

        s = shm;

        fork();
        fork();
        fork();
        fork();
        fork();
        fork();
        fork();
        fork();
        fork();
        fork();
        fork();
        fork();

        int i = 0;
        while(i < 10){
                *s = *s+1;
                printf("\nValor de variable en restador = %d",*s);
                //sleep(4);
                i++;
        }
}
```

```c
#mostrar_variable.c
#define TAMANIO 1

void main(){

        int shmid; // ID del segmento de memoria compartida
        key_t key;
        int *shm, *s;

        key = 56789;  // Nombramos el segmento de memoria compartida

        // Creamos el segmento
        if( (shmid = shmget(key, TAMANIO, IPC_CREAT|0666)) < 0){
                perror("Error al crear el segmento de memoria");
                exit(1);
        }

        // Ligamos el segmento al espacio de direcciones
        if( (shm = shmat(shmid, NULL, 0)) == (int *)-1){
                perror("Error al ligar la memoria");
                exit(1);
        }


        s = shm;

        int i = 0;
        while(i < 1000){
                printf("\nValor de variable actual = %d",*s);
                sleep(1);
                i++;
        }
}
```

Al ejecutar los dos programas ejecutables, obtenidos al compilar restador.c y sumador.c, esperaríamos que el resultado sea cero, ya que cada uno de estos procesos suma o resta exactamente la misma cantidad de unidades. Sin embargo, podemos ver que el resultado de la variable compartida será diferente a cero cuando dichos procesos finalizan. Esto debido a que están en una *condición de carrera*. Por otro lado, las instrucciones *fork()* permiten crear muchos procesos sumadores y restadores, cada uno de los cuales sumará o restará exactamente los mismos valores. Al tener demasiados procesos ejecutándose al mismo tiempo forzamos un error en el resultado final de la variable compartida. De esta manera podemos observar el problema de la condición de carrera (*race condition*).

Referencia: Principles of operating systems – Naresh Chauhan

Revisar también: Operating systems, Design and Implementation sección 2.2.1

Revisar también: Operating systems, Concepts and Techniques sección 8.1.1

Problema productor/consumidor

Al interactuar los procesos pueden necesitar un dato o resultado (de entrada) de otro proceso. En este caso el proceso que necesita de otro debe esperar hasta que reciba la entrada del otro proceso, es decir, un proceso depende del otro. Para obtener los resultados deseados es necesario tener un control sobre el proceso de tal manera que podamos obligarlo a esperar a que la ejecución del otro se haya completado.

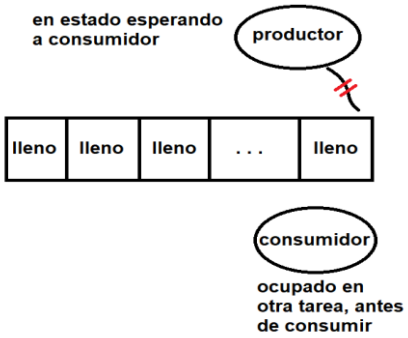
Ejemplo. Se tiene un búfer que es mantenido por dos procesos. Uno de estos es llamado ***productor*** y produce datos que coloca en el búfer. El otro es llamado ***consumidor*** y necesita datos del búfer, los cuales, consume. Para esta aplicación se necesita coordinación y sincronización ya que estos procesos son dependientes uno del otro. Si el búfer está vacío el consumidor no debería intentar consumir nada del búfer. Por otro lado, si el búfer está lleno no se deberían producir nuevos datos.

Para dar seguimiento del búfer para determinar si está vacío o lleno, usamos una variable contador que contiene el número de elementos en el búfer. Esta variable debe ser compartida y actualizada por ambos procesos.

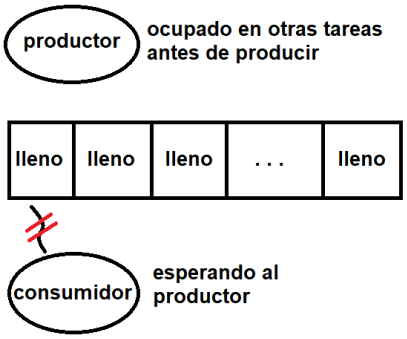
El consumidor debe verificar el valor de contador antes de consumir un elemento del búfer. Si $contador \geq 1$ significa que hay uno o más elementos en el búfer, por lo tanto, puede consumir. Después de verificar que hay elementos en el búfer, consume y actualiza el contador: $contador - -$.

Por otro lado, el productor también debe verificar el contador antes de producir datos y agregarlos al búfer. Si el contador es menor que el tamaño máximo del búfer significa que hay espacio para añadir más datos. De ser así produce y agrega un elemento al búfer y actualiza el valor de contador: $contador + +$.

Sea MAX el tamaño máximo del arreglo del búfer, puede suceder que $contador == MAX$ y el consumidor esté ocupado ejecutando otras instrucciones o que aún no se le haya asignado su intervalo de tiempo. También en ese momento el productor puede estar listo para seguir produciendo, pero ya que el búfer está lleno deberá esperar a que el consumidor termine de consumir el siguiente elemento y actualice el contador $contador = MAX - 1$.



Otra situación sucede cuando el búfer está vacío, es decir, $contador == 0$ y el productor está ocupado ejecutando otras tareas o no se le ha asignado tiempo. Ya que el búfer está vacío el consumidor necesita esperar a que el productor produzca un dato y lo agregue al búfer.



En los casos anteriores los procesos involucrados están interactuando y comunicándose a través de una variable compartida (comunicación indirecta por medio de memoria compartida). Dicha variable es necesaria para tener una ejecución sincronizada de los procesos.

Por otro lado, hay casos en que los procesos necesitan compartir datos que no son necesarios para sincronizar el acceso o controlar esa sincronización, sino para fines de lectura. En este caso, no es necesario mantener datos compartidos ya que afecta al costo de acceso. Así que los procesos también pueden comunicarse por medio de mensajes y ser conscientes de la existencia del otro. Este tipo de comunicación entre los procesos se conoce como ***paso de mensajes*** y es usado donde no es necesaria (o no es posible) la comunicación por medio de memoria compartida. Por ejemplo, los sistemas distribuidos, ya que los procesos residen en diferentes nodos. El paso de mensajes es un sistema en el cual los procesos conocen a los otros explícitamente e intercambian mensajes a través de un sistema de llamadas. Un sistema de llamadas se usa para enviar mensajes a otros procesos y recibirlos. Cada mensaje tiene un formato fijo que consiste en el mensaje y del nombre de su remitente (o receptor). En este tipo de comunicación entre procesos también es necesaria la sincronización. Cuando un remitente envía un mensaje, no es necesario que el receptor esté listo para recibirlo. En este caso, el remitente será bloqueado y el mensaje será copiado a un búfer. Se activará solamente cuando el receptor objetivo ejecute su llamada al sistema para recibir mensajes. Después el mensaje que se encuentra en el búfer es enviado al proceso receptor. De manera similar, cuando el proceso esté listo para recibir un mensaje, no necesariamente el remitente lo estará para enviar un mensaje. En este caso el receptor será bloqueado y será activado solamente cuando el remitente tenga la intención de enviar el mensaje.

Ver ejemplo 7.3 y sección 7.2.4 del libro Principles of operating systems de Naresh Chauan

Sección crítica y exclusión mutua

El sistema operativo debe ser capaz de asegurar la cooperación de los procesos, que tienen diferente velocidad de ejecución, sin que surjan problemas. Si más de un proceso comparte un dato, entonces los procesos deben ser sincronizados para evitar accesos múltiples a los datos compartidos. El requerimiento para que dos o más procesos no accedan a los recursos compartidos al mismo tiempo es llamada ***exclusión mutua.*** Cumpliendo con ella evitamos que los procesos no accedan o actualicen los datos compartidos al mismo tiempo. La exclusión mutua requiere que el código que accede y modifica los recursos compartidos sea protegido, y que no sea ejecutado por los procesos de manera simultánea. La sección de código que accede a los recursos compartidos es llamada ***región crítica (RC).*** Se debe definir un criterio que cada proceso debe pasar para entrar y ejecutar su región crítica, a este se le llamada: ***criterio de entrada.*** La exclusión mutua (MUTEX) también requiere que cuando un proceso ha terminado su ejecución los otros procesos que esperan deban ser informados que pueden entrar a su correspondiente sección crítica: ***criterio de salida.***

```
1  proceso p(){
2    haz{
3      :
4      criterio de entrada
5      :                      Sección crítica
6      :
7      criterio de salida
8      :
9    }mientras(...)
10 }
```

Para implementar el protocolo de exclusión mutua se necesita hacer un seguimiento a todos los procesos que deseen entrar a su región crítica. Si hay algunos procesos en espera (p.e. en una cola) y ningún proceso está ejecutando su sección crítica, entonces solo los procesos en espera podrán tener permiso para ejecutar su sección crítica. Es decir, no se va a reservar permiso para aquellos procesos que no estén ejecutando su sección crítica. La solución para problemas de región crítica debe satisfacer las siguientes características:

- **Exclusión mutua.** El protocolo permite que más de un proceso actualice al mismo tiempo un dato global .
- **Progreso.** La implementación de este protocolo requiere que se mantenga un seguimiento de todos los procesos que deseen entrar a su región crítica. Si hay algunos procesos en la cola de espera, y actualmente no hay ningún proceso que esté ejecutando su región crítica, entonces solo se debe otorgar permiso a los procesos en espera para participar en la ejecución. Es decir, no se va a reservar permiso a ningún proceso que no se encuentre en la región crítica.
- **Espera limitada.** Un proceso que se encuentre en su región crítica estará ahí solo por un tiempo finito ya que la ejecución no toma mucho tiempo, por lo tanto, el tiempo en la cola de espera no será muy largo.
- **No interbloqueo (no deadlock).** Si un proceso está ejecutando su región crítica y hay procesos en la cola de espera pero que no se les ha asignado el permiso para entrar a la suya, entonces puede suceder un interbloqueo en el sistema. Por lo tanto, el protocolo debe asegurar que todos los procesos en espera tengan asignado su permiso para entrar a la región crítica.

Interbloqueo. En un entorno multiprogramación (donde dos o más procesos se pueden alojar en memoria y se ejecutados concurrentemente) muchos procesos compiten por una cantidad finita de recursos. Cuando un proceso requiere recursos y estos no están disponibles en ese momento entra a un estado de espera. Los procesos en espera pueden permanecer así indefinidamente si los recursos que solicitan están ocupados por otros procesos. Esta situación es llamada ***deadlock o interbloqueo.***

Solución para implementar sección crítica para dos procesos

Solución 1. Supongamos que tenemos dos procesos p_1 y p_2, y queremos implementar sincronización entre ellos. Cada proceso tiene una sección crítica y necesita esperar al otro cuando está utilizando su RC. Una primera solución implementada utiliza una variable compartida $turno$ para asegurar la exclusión mutua. Con esta variable se indica cuál proceso tiene permiso para acceder a su sección crítica. El criterio de entrada en este caso es el ciclo $mientras$. p_1 y p_2 no pueden entrar a su sección crítica hasta que la variable $turno$ sea igual a 1 o 2 respectivamente. Si $turno = 1$, entonces p_1 tendrá permiso para entrar y ejecutar su sección crítica. Al mismo tiempo, si p_2 está listo para entrar a su sección crítica no tendrá garantizado su permiso para ejecutar su sección crítica. Cuando p_1 termine la ejecución de su RC entonces $turno = 2$ y entonces p_2 podrá ejecutar su RC. Como podemos ver esto garantiza la exclusión mutua.

```
proceso p1{                          proceso p2{
  int turno;                           int turno;
  turno = 1;
  haz{                                 haz{
    mientras(turno != 1);                mientras(turno != 2);
    RC                                   RC
    turno = 2;                           turno = 1;
    ...                                  ...
    ...                                  ...
  }mientras(verdadero);                }mientras(verdadero);
  ...                                  ...
}                                    }
```

Ahora supongamos que p_1 está en su RC y p_2 no. Cuando p_1 salga de su RC asigna el valor $turno = 2$. Sin embargo, puede ser que p_2 no necesite aún ejecutar su RC y que p_1 regrese otra vez a esperar su turno. Esto viola el protocolo mencionado anteriormente porque no es posible observar el progreso (no hay seguimiento de los procesos que están a la espera de entrar a su RC). El problema en este caso se debe a que no se guarda el estado de cada proceso para determinar si está en su región crítica o está en espera de entrar en ella.

Solución 2. Para almacenar el estado de cada proceso podemos usar dos variables: $f1$ y $f2$, estas guardarán el estado de los procesos p_1 y p_2 respectivamente. Si un proceso entra a su RC entonces asignará el valor cero a su correspondiente bandera y cuando salga la cambiará a uno. Esto indica que si un proceso está actualmente usando su región crítica, entonces el otro proceso deberá esperar hasta que la bandera cambie a uno, es decir, la región crítica está disponible. Así, estas variables o banderas eliminan el problema de progreso observado en la primer solución mencionada anteriormente.

```
proceso p1{
   int f1, f2;
   f1 = 1;
   haz{
      mientras(f2 != 1);
      f1 = 0;
      RC
      f1 = 1;
      ...
      ...
   }mientras(verdadero);
   ...
}
```

```
proceso p2{
   int f1, f2;
   f2 = 1;
   haz{
      mientras(f1 != 1);
      f2 = 0;
      RC
      f2 = 1;
      ...
      ...
   }mientras(verdadero);
   ...
}
```

Por otro lado, esta solución tiene otros problemas. Al inicio, cuando ningún proceso está en ejecución, tango p_1 como p_2 intentarán entrar a su región crítica ya que los dos tienen sus banderas con valor uno. Esto viola la exclusión mutua. Para evitar esto, podemos asignar el valor de cero a cada una de las banderas antes del ciclo $mientras$. Esto resolverá el problema de la exclusión mutua pero aún habrá otro más. Supongamos, que en el momento en que p_1 inicia y ejecuta la asignación $f1 = 0$, $p2$ interrumpe y se le asigna la ejecución. Así que p_2 ejecuta la asignación $f2 = 0$. En esta situación, si p_2 continua y ejecuta su ciclo $mientras$, entonces no será capaz de continuar ya que estará esperando a que la bandera de p_1 cambie a $f1 = 1$. De manera similar, si se le asigna la ejecución a p_1 y ejecuta su ciclo $mientras$ tampoco será capaz de continuar ya que estará esperando por p_2. Como ambos procesos están esperando el uno al otro se causa un $deadlock$ en el sistema.

Solución 3. El $deadlock$ de la solución anterior se puede eliminar si cada proceso, antes de entrar a la sección crítica, verifica si el otro ya está listo o no. El proceso p_1 verifica si $f2 == 0$ (¿p_2, estás ocupando tu sección crítica?) y entonces asigna a su bandera $f1 = 1$. De igual manera, p_2 verifica si $f1 == 0$ y entonces asigna a la suya $f2 = 1$. Esta solución elimina el $deadlock$ pero causa otro problema llamado $livelock$ (tarea: investigar).

```
proceso p1{
   int f1, f2;
   f1 = 1;
   haz{
      f1 = 0;
      si(f2 == 0)
         f1 = 1;
      mientras(f2 != 1);
      RC
      f1 = 1;
      ...
      ...
   }mientras(verdadero);
   ...
}
```

```
proceso p2{
   int f1, f2;
   f2 = 1;
   haz{
      f2 = 0;
      si(f1 == 0)
         f2 = 1;
      mientras(f1 != 1);
      RC
      f2 = 1;
      ...
      ...
   }mientras(verdadero);
   ...
}
```

Solución de $Dekker$. Si se toma en cuenta tanto el turno y el estado de cada estado el algoritmo no tendrá los problemas mencionados anteriormente. La solución que toma en cuenta esto fue dada por $Dekker$. Dicha solución satisface todos los criterios del protocolo para solucionar los problemas de región crítica: exclusión mutua, progreso, espera limitada y $deadlock$. De manera que ambos procesos no intentarán entrar de manera simultánea a su RC debido a la nueva variable $turno$. Si p_1 comienza primero y encuentra que $f1 == 0$, es decir, p_2 desea entrar a la RC entonces se le permitirá entrar a p_2 solo cuando $turno == 2$. En otro caso, tendrá que esperar a que $f2 == 1$ y así p_1 podrá entrar a su RC. Si se asigna inicialmente $f2 = 1$, entonces p_1 se saltará el ciclo $mientras$ e irá directamente a su RC. Esto evitará tanto el $deadlock$ como el $livelock$.

```
proceso p1{
   int f1, f2, turno;
   f1 = 1;
   haz{
      f1 = 0;
      mientras(f2 != 1){
         si(turno == 2){
            f1 = 1;
            mientras(turno == 2);
            f1 = 0;
         }
      }
      RC
      turno = 2;
      f1 = 1;
      ...
   }mientras(verdadero);
}
```

```
proceso p2{
   int f1, f2, turno;
   f2 = 1;
   haz{
      f2 = 0;
      mientras(f1 != 1){
         si(turno == 1){
            p2 = 1;
            mientras(turno == 1);
            f2 = 0;
         }
      }
      RC
      turno = 1;
      f2 = 1;
      ...
   }mientras(verdadero);
}
```

Semáforos

Como hemos mencionado anteriormente el problema de condición de carrera es evitado cuando una variable global no es actualizada por más de un proceso a la vez. Las operaciones que no pueden ser superpuestas o intercaladas con cualquier otra operación se conocen como $operaciones\ indivisibles\ o\ atómicas$. Esto significa que si la operación es indivisible, entonces no tendremos problemas de condición de carrera.

El semáforo es una herramienta muy popular usada para sincronizar procesos. También es usado para proteger cualquier recurso como datos globales en memoria compartida que necesitan ser accedidos y actualizados por muchos procesos de manera simultánea. El semáforo actúa como un guardia o una cerradura sobre el recurso compartido. Siempre que un proceso necesita acceder al recurso primero necesita el permiso del semáforo. Si el recurso está libre (es decir, si ningún otro proceso está actualizando o accediendo al recurso) entonces se le dará permiso al proceso en otro caso se le niega. Cuando se le niega el permiso, el proceso que pidió el acceso necesita esperar hasta que el semáforo se lo permita.

Un ejemplo de la vida real. Un guardia está sentado fuera de un salón. Siempre que un profesor llega para impartir clase le pide permiso al guardia para entrar al salón. El guardia le permitirá al profesor la entrada al salón si el salón está libre, pero en otro caso se le lo niega. Como podemos ver el salón está siendo usado en exclusión mutua con ayuda del guardia. Los semáforos actúan de manera similar.

El semáforo se implementa como una variable entera S, la cual, puede ser inicializada con cualquier valor entero positivo. El semáforo se implementa como una variable entera S que puede ser inicializada con cualquier número entero positivo. Para acceder al semáforo se utilizan dos operaciones indivisibles: $wait$ y $signal$, denotadas como P y V respectivamente.

Operación wait

```
P(S){
   mientras(S <= 0);
   S = S - 1;
}
```

Operación signal

```
V(S){
   S = S + 1;
}
```

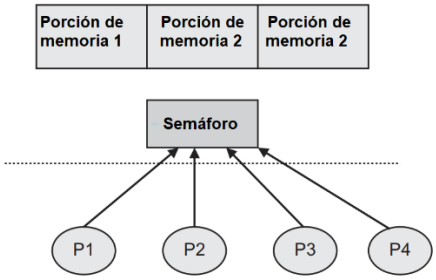
La implementación de un semáforo protegiendo la RC se realiza de la siguiente manera. Siempre que un proceso quiera ingresar a la RC necesita ejecutar una operación $wait$. Esta operación es la condición de entrada. Si la RC está libre entonces se le permite el acceso en otro caso se le negará. El valor del semáforo se disminuye en 1 cuando un proceso accede a la RC. Observemos que el valor del semáforo nos indica la disponibilidad de la RC. Inicialmente el valor del semáforo es 1, pero si un proceso accede entonces al disminuir ese valor será 0. Por otro lado, si otro proceso intenta acceder a la RC. Entonces no se le va a permitir la entrada hasta que el valor del semáforo sea mayor que cero.

Cuando un proceso sale de la RC, ejecuta una operación $signal$. Este será el criterio de salida. De esta manera, la solución a la RC usando el semáforo satisface el protocolo diseñado. El semáforo cuyo valor será cero o uno es conocido como semáforo binario. Un recurso que tiene una sola instancia y que es usada en exclusión mutua puede usar un semáforo binario para sincronización.

```
...
haz{
   wait(semáforo){


   }
   signal(semáforo)
   ...
   ...
}mientras(verdadero);
```

En general un semáforo puede tomar cualquier valor positivo. Por ejemplo, si tenemos tres porciones de memoria que necesitan ser actualizadas en exclusión mutua, entonces el semáforo usado para protegerlas tomara el valor 3. Esto significa que hay tres procesos que acceden al semáforo. Después de dar acceso al tercer proceso, el valor del semáforo se vuelve cero, así que no se le permite el acceso a cualquier otro proceso a menos que el proceso actual salga de la RC. Este tipo de semáforo se conocer como $counting\ semaphore$.



Existe otro tipo de semáforo binario llamado $mutex$. En un semáforo binario la RC bloqueada por un proceso puede ser desbloqueada por cualquier otro proceso. Pero con $mutex$ solo el proceso que bloquea la RC puede desbloquearla. Un problema al implementar este tipo de semáforos es que cuando un proceso no tiene permiso de acceder a la RC se cicla al esperar dicho permiso. Esto no produce resultados y consume ciclos de CPU por lo que se desperdicia tiempo de procesador. El tipo de semáforo que hace esto se le llama $spinlock$, ya que el proceso dará vueltas $(spin)$ esperando obtener permiso para acceder a la RC. Para ahorrar tiempo de procesador, el proceso que no es capaz de acceder al recurso compartido debe ser bloqueado. Ya que puede haber varios procesos en espera es necesario almacenarlos en una cola de espera. Un proceso se "despierta" de la cola cuando otro proceso libera el recurso y el semáforo le permite entrar a su RC. Aunque esta solución implica un gasto en el cambio de contexto (operación de desalojar un proceso del CPU y reanudar otro). El siguiente pseudocódigo representa la implementación de este semáforo.

```
Struct {
        int sem_value;
        queue sem_q;
} semaphore;

semaphore S;

Operation Wait

P(S)
{
    S.sem_value = S.sem_value –1;
    if (S.sem_value < 0)
    {
        Add the process to S.sem_queue;
        Block the process;
    }
}


Operation Signal

V(S)
{
     S.sem_value = S.sem_value + 1;
     if (S.sem_value <= 0)
     {
         Remove the process from S.sem_queue;
         Wakeup the process;
     }
}
```